5. THE TRANSPORT LAYER -
   Layer 4

Read Chapter 6.0 - 6.5 of [Tannenbaum96]. Skim 6.6.

The transport layer is in some ways the "heart" of your data communication system. It is your agent for reliable communication with the quality and speed of service that you request (your request may be denied but at least you are informed your desired quality and speed are not available).

The transport layer runs in your computer, while the network layer runs in a network's local or remote nodes.

Note that if the Packet Handler is very close to (or a board inside) your computer, you could potentially just use RS-232 (or your hardware bus) to transfer data to/from the Packet Handler. This would be done if you felt the RS-232 or bus were reliable enough to not need layers 1-3 inside your computer and in the Packet Handler.

Note: There is a number of references in this section to man pages for socket programming. Please print out all the man pages by following the instructions in Section 5.6.6 of the notes.

5.1 Transport Layer Services

The transport layer "bridges the gap" between the services needed by session layer, and the possibly unreliable connectionless service provided by the network. Note that if the network layer is considered very reliable and offers the types of service needed (connections, multiplexing to reduce costs, speed, etc.), then the transport layer is not needed.

5.1.1 Handling Lost/Duplicated Packets

But if you don't trust B.C. Tel's DATAPAC (X.25) service to not ever:

a) Lose a packet crediting your bank account.

b) Duplicate a packet debiting your bank account.

- then you need end computer to end computer checking done by your transport layers. "Ever" means not just during normal operation but during power outages, lightning strikes, downloading of new network software onto nodes and re-booting them while the net is running, etc. Note that we are not talking about power failure at your source and destination computers, but at inner network nodes. When inner network nodes re-boot, your computers at each end may be alive and ready to accept duplicate packets!

Of course if one of your end sites does fail, it would be nice if the transport or session layers could make a graceful recovery and resume where they left off!

5.1.2 Expedited Delivery

Sometimes later data may supercede and need to interrupt processing (in the net or in the destination application) of earlier sent data. The simplest example is a terminal application needing to send a break, escape, or control-C. Rather than just giving this class of message higher priority (via a higher percentage of throughput), an attempt is made to persuade the network or destination application to let this expedited data jump ahead in the queue. If the network is exhibiting flow back pressure, the transport layer could even attempt to open a new connection to flash the "break" through. (We'll talk more about multiple net connections later).

5.1.3 Upward Multiplexing

Often there are several users/tasks running on your computer, that each need data communication out of your computer via it's only connection. Also, networks may allow multiple virtual circuits (connections) through a single network physical connection to allow several applications to share the cost of a single (say 56kbps) link into your site, but often still charge per minute of each virtual connection. So, the transport
layer purely for the purpose of avoiding this cost, can multiplex several "transport" connections onto one net connection (if all the transport connections are going to the same destination). This is called **upward multiplexing**.

When are the network providers going to start charging only for # of bytes sent, rather than for time? Admittedly, connections often cost the net service provider for reserved buffers along the route. But the only value customers are not willing to try and bypass is transport of bytes, and possibly peak throughput available (e.g. 2.4 vs. 56 kbps).

### 5.1.4 Transport Endpoint Addressing

Transport addresses are needed, as often a program can have several connections to several other computers.

### 5.1.5 Quality of Service

In connection-oriented transport layers, during connection set-up, you can ask for particular levels of:

a) Residual error rate guarantee (FCS length).
b) Throughput needed
c) Response time needed.
d) Priority.
e) Security.
f) Transport failure probability guarantee.
g) Connection release delay and failure probability.

A Q.O.S. proposal is made during transport connection set-up; it is either accepted, denied, or a counter-proposal is returned! This latter case is called QOS negotiation.

### 5.1.6 Downward Multiplexing

Sometimes, B.C. Tel may bring a high capacity (56 kbps) physical connection into your premises, but only allow you 2400 bps per virtual connection. If the session layer requests 9600 bps throughput is needed, the transport layer could do **downward multiplexing**.

But after downward multiplexing, the destination transport layer must re-sequence and re-assemble the transport packets, which may have:

a) Been broken in quarters, or
b) Had every 4th one sent on a given net connection

The other computers when sending packets to the first, must specify which of the connections to the program in the first computer they are sending to.

### 5.2 Transport Difficulties

A full featured transport layer is very complex, due to the difficulties introduced by:

1) An underlying unreliable datagram net.
2) Having to provide reliable connection establishment, failure recovery, and disconnect under **all** circumstances (There are weird circumstances possible!).
3) Multiplexing (both upward and downward).

#### 5.2.1 Error Control

Normally TPDUs arrive without bit errors due to the abilities of lower layers. But due to a variety of possibilities (e.g. power failure at intermediate nodes), it is possible to lose or receive duplicate packets. **Thus sequence numbers are required at yet another layer!** (Note: They are not used in all net layers. e.g. Datagram nets).

Sequence numbers are also required for downward multiplexing, both for error control of the individual sub-streams, and for re-sequence/re-assembly of the original TPDUs.
5.2.2 Flow Control

Transport layer flow control is needed for two reasons:

1) Upward multiplexing: needed for same reason as net layers. Multiple users don’t want to be back-pressured just because one needs to be.

2) To help prevent flooding of a datagram network. Transport layer flow control essentially only allows the maximum transport window size # of packets into the network at any given time.

Note: Flow control may be needed because either transport entity (finite state machine and TPDU formatting/re-assembly) can’t handle the rate, or because the destination application itself can’t handle the rate (CPU or I/O bound).

Transport flow control implementation options:

a) TPDU discarding with source time-out re-send.

b) Back pressure net and let it handle it (poor for upward multiplexing).

c) Sliding window.

d) Credit allocation scheme - De-couples ACK mechanism from flow control mechanism. Allows continuous adjustment of max. window size. Needs two fields in TPDU (e.g. “Ack 2 + Credit 3” means Ack 2 and "currently" willing to accept 3, 4, + 5).

5.3 Transport Connection Management

Transport connections are made and broken by sending or receiving control TPDU. These are discussed in the context of requests, indications, responses, and confirmations.

The ISO control TPDU service interfaces are tabulated in Table 12.4 of [Stallings 94] as:

- T-Connect. request(called address, ..., QOS requested..)
- T-Connect.indication()
- T-Connect.response(QOS available)
- T-Connect.confirm(QOS granted)
- T-Data.request(data)
- T-Data.indication(data)
- T-Expedited-Data.request(data)
- T-Expedited-Data.indication(data)
- T-Disconnect.request()
- T-Disconnect.indication(reason,....)

Examples of the application of control TPDU operations in various scenarios can optionally be seen in Figure 12.13 of [Stallings 94].
Strange scenarios, like two transport user applications both trying to initiate a connection at the same time, can happen. It is thus best to handle the various cases by a rigorously designed finite state machine running in each end’s transport provider task.

See for example Figure 12.6 of [Stallings 94] for the “two-way handshake” (i.e. simple) F.S.M. This FSM has several interesting aspects:

- It only covers connection + release. Data transfer would be handled sub-states within the open state.
- The two idle states are the same. Two are shown simply to unclutter the diagram.
- Can you identify where in the diagram the Sun T.L.I. calls and returns would fit? Note: RFC means Request For Connection packet type. CLS means Close (i.e. the packet type associated with a t_sndrel() call).
- This FSM will handle simultaneous connection initiation or release initiation, though not extremely reliable (cf. “three way handshake”).

It is very hard to build a reliable, connection-oriented transport layer on top of an unreliable datagram net; harder than any other datacom problem. This is due to interaction of the problems involved in:

1) Duplicate handling.
2) Connection establishment.
3) Connection termination.
4) Crash recovery.

Each solution to a particular difficulty you think up seems to introduce other problems.

5.3.1 Duplicate Handling

Duplicate data frames are created by retransmission resulting due to delayed frames or lost ACKs. If a net or transport duplicate gets caught in datagram net congestion it may re-surface minutes later! This causes serious problems:

**Problem A**) The modulo sequence numbering may have “wrapped” and an old duplicate may be “accepted” as a valid new packet.

(Note: This does not happen at the data link layer or in a virtual circuit implemented net layer because they act as a FIFO queue. But it can happen when...
implementing a connection-oriented transport (or net) interface on top of a datagram net implementation.)

Solutions to Problem A - Either:
• use very large modulo sequence space ($2^{32}$),
• or time stamp packets and discard very old ones.

Problem B) Another problem caused by duplicates it that a connection may close, then immediately be re-opened. An old duplicate (say packet #3) from the first connection could have been delayed and re-surface during the second connection, and be accepted as correct!

Solutions to Problem B (all of them pretty ugly):
• Use one sequence space across multiple instances of a connection. When re-starting a new connection to the same other station, start sequence numbering where you last left off, rather than at 0. Disadvantage: Have to remember where you left off with every other station in the world. (I told you they were ugly!)
• Use different transport addresses for each connection. But then the concept of “well known” transport addresses (for things like ftp servers) fail.

Either of the above fail with a re-boot anyway.
• Wait at least a minute between similarly addressed connections --- slow/delay.

Best Solution: Time stamp packets and don't accept any from same endpoint and station address before the connection opening time.

5.3.2 Connection Establishment

Connection establishment using two-way handshake can be fooled by an RFC (Request For Connection) control packet which is a duplicate and which is delayed excessively by a network node. This triple combination is rare, but locks up a particular ‘transport user-to-transport user’ address pair.

Transport Endpoint A    Transport Endpoint B

RFC

timeout RFC

RFC

Frame 0

Close

Close

Delayed RFC

`A' thinks it is getting a request for a second connection, so sends an RFC as a reply.

`B' gets an RFC and responds to what it thinks is a request for a new connection.

Normally, if a second connection between two specific transport addresses is requested when one is already in existence, it is refused. But the problem in the above diagram is that there is no way to tell that the delayed duplicate RFC is not a valid new request.

The key to solving this problem is four fold:
1) We must distinguish between an initial RFC and one which is a ‘reply’ RFC.
2) We must sequence number control packets too!
3) We must acknowledge a reply RFC (3 way handshake).
4) We need a reset (RST) packet type, to tell the other end something has gone wrong and to start again.

A 3 way connection normally works like this:

```
A                  B
RFC 23

RFC 7', ACK 24

I(24)P(8')
```

Numbered 3 way handshakes aids in the following 2 ways:
1) Delayed Dupl. RFC 18'.

‘A’ must respond to what it thinks is a request for new connection.

RFC 23, ACK 19'.

B rejects this as the ACK 19' implies B initiated this new connection (and it knows it didn’t!).

The ‘Reset’ tells ‘A’ that this was not the start of a real connection.

2) ‘A’ rejects delayed dupl. RFC 19', ACK 74

Reset 19'

Reset 23

Note 1: A port always discards an RFC for a connection from another port it already has a current connection to.

Note 2: It is possible that two stations could both try to connect to each other at the same time, with their requests crossing in transit. In some implementations this will result in a 2x3=6 way interaction. Or with some careful analysis of the finite state machines involved, may reduced to 4 way.

Note 3: What do you use as the first sequence #?

Anything that will prevent or reduce probability of confusion from an old duplicate RFC re-surfacing. It should be different from the RFC used on your last few connections. e.g.

i) 1 + sequence # at end of last connection, or  

ii) Time stamp (No need for net sync of time! why?),  

iii) or 32 bit random number.

See Figure 6-28 of [Tannenbaum96] for a three way handshake finite state machine.

5.3.3 Connection Close (Disconnect)

In order to deal with lost or late surfacing duplicates of release connection requests, it is wise to use:

a) 3 way close handshake - each side must explicitly acknowledge the release request of the other.

b) Even though Close and Ack_CLOSE are not data packets, they must be sequence numbered to prevent confusion during lost or duplicate control packets.

- Close is numbered with the usual next sequence number.

- Ack_CLOSE echoes that sequence number back to the close initiator.

c) If no Ack_CLOSE received, it may have been lost. Re-try sending the Close several times, then give up!

Note: Might not need all three if use 32 bit random start numbers for data packets.
Note: If close is simultaneously initiated from both ends, it seems like a 4 way handshake.

Note: If close is simultaneously initiated from both ends, it seems like a 4 way handshake.

A

\[ \text{CLS 8} \]

B

\[ \text{CLS 5'} \]

B knows it wants to close.

A

\[ \text{Ack_CLS 8} \]

B now also knows A wants to close.

B knows that A knows that B wants to close!

5.3.4 Transport CPU Crash

- If there is no response from other end, after an appropriate timeout interval, sent a transport reset (RST) TPDU.
- This will reset the other end, which may have crashed and re-booted not knowing it dropped a connection.
- If no session transaction record logged on disks, this will likely cause some transport state and data to be lost.

5.4 Examples Of Transport Protocols

5.4.1 TP4 (ISO 8072-8073)

Some recent public networks created by telecom companies and governments use this connection-oriented transport service defined by the ISO. ISO considered what features a transport layer must provide on top of various degrees of underlying network (un)reliability.

Type A1 Underlying Net

A type A1 underlying network offers a minimal error rate, few network failures/resets, and proper sequencing. (e.g. X.25 Datapac)

A transport layer sitting above must provide flow control (via sequence numbers) only if upward multiplexing is needed, and should use a simple 3 way handshake (control packets needn’t be numbered).

Type A2 Underlying Net

A type A2 underlying net offers a minimal error rate, few net resets, but is non-sequencing.

A transport layer sitting above an A2 net must use sequence numbers for re-sequencing, and must use sophisticated 3 way handshakes (i.e. sequence numbered control TPDUs). Also needs flow control if upward multiplexing.

Type A3 Underlying Net

A type A3 underlying net offers a minimal error rate, few net resets, is non-sequencing, and has an annoyingly small maximum packet size (e.g. to combat mobile radio signal strength picket fencing)

A transport layer sitting on top of an A3 net must have sequence numbers on all TPDUs (as above), but if not stream-oriented (TCP is, ISO 8073 is not) must also have some kind of "end-of-transport-block" capability. This tells the receiving transport layer that the previous 5 (say) TPDUs were really passed from the session layer as one block, but had to be broken up to fit on the net. The receiving transport layer will re-assemble them, and pass them up to the session layer at the destination as one block.

Type B Underlying Network

A type B underlying net has a minimal packet error rate (when it is running), but is plagued by frequent failures/resets which are properly brought to the attention of the transport layer.

A transport layer sitting on top of a B net must, in addition to the requirements for Type A2 above, be able to do the following:
a) On receiving an X.25 Reset Notification -- send a special control TPDU alerting the other end that it has received a reset, repeat its last Transport ACK indicating up to which transport receive sequence # it received correctly, then wait for a similar control TPDU from the other end. Note: a net reset does not mean the (virtual) connection was lost and needs to be re-set-up.

b) On receiving an X.25 net Restart Notification implying even the net connection was lost -- the original originating transport entity must request a new net virtual connection, and then send a control TPDU to the original answering transport entity explaining that this new virtual circuit ID should be mapped to the old transport connection ID that we are trying to get running again from where it left off. Then goto to a).

**Type C Underlying Network**

A type C underlying network is error prone in all ways, and non-sequencing.

A transport layer sitting on top of a type C net must have all features discussed so far. Quality of service negotiation may even, in extreme cases, request a checksum to detect bit errors.

ISO decided that, rather than defining 10 classes of transport layer (for 5 network types, with or without multiplexing), that it would only define standards for the 5 most frequent classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>Underlying Net</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>Simple class</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>Basic (net) error recovery class</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>(Simple) multiplexing class</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>Error Recov. + mux. class</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
<td>Error Detection + Recov. class</td>
</tr>
</tbody>
</table>

Class 0 and 1 are so simple that they don’t even have flow control.

The last class, 4, is widely referred to as "TP4".

[Stalling 94] Table 12.3 shows the features defined in ISU 8072/8073 for each class. Almost everyone uses TP4, unless performance is a critical issue.

Optional Table 12.3 of [Stallings94].

**NOTE:**

The ISO version of TP4 does not use a 3 way handshake for connection close. Instead it uses "abrupt termination", which could lose data if it is delayed and re-surfaces after a close. The American National Standards Institute was so disgusted with this oversight that ANSI added a whole extra TPDU type to avoid this. So ISO TP4 and ANSI TP4 standards are not identical and thus may not understand each other’s connection closes!
5.5 The Transport Software Interface

The transport interface is the boundary between the user’s (session) process, and the transport provider process on the CPU.

But on a given CPU, there may be several user tasks using the provider at the same time. Or one user task may open several connections to different destinations.

As shown above, there may even be several transport providers on a given CPU to provide different protocol connections!

5.5.1 Transport Addresses

Transport addresses are more specific that just the network address of the machine. In particular, it must indicate a particular I/O stream in the CPU.

A transport address is composed of 4 parts:

i) The network address “Domain” - each major network has it’s own addressing format. You must specify which scheme (e.g. for internet address format specify the constant AF_INET from <sys/types.h>).

ii) The CPU’s network address within the domain, specified using the correct format.

iii) The transport protocol - i.e. which transport provider task within a given CPU.

iv) A ‘port’ or ‘end point’ (integer) - which is unique to that end point on that provider task on that CPU.

5.5.2 Transport Communication Phases

Communication can be broken down into a number of phases. Some of these phases are similar to those used in file I/O. Certain procedure calls are used during specific phases.

a) Local management phase - specify a provider and allocate/de-allocate a port, etc. The is like opening a file in Unix, and being returned an integer I/O descriptor index for the open file/socket.

b) Connection phase - optional; used only for connection-oriented communication. Specifies full address (address format + machine_address + port_number) of server’s advertised socket to connect to.

c) Data transfer phase.

d) Connection release phase - optional.

e) Continued local management - close the socket/file

Note:

Local management functions may actually be called for various purposes throughout the other phases (i.e. throughput not high enough; then request higher priority for the remaining duration of connection).

5.6 Tutorial On SUN Sockets

5.6.1 Background

Sun provides an application programmer interface (API) for the Transport layer services specified in the ISO 8072 transport service definition. This is called Transport Level Interface (TLI), and has been used in the past for Cmpt 371 programming assignments. This is because it was thought that the ISO versions of protocols and services would become popular. Even Sun’s own documentation suggested that all future programs using transport layer services should be written using TLI rather than the older Berkeley UNIX ‘socket’ API. In addition, in the old SUN OS 4.1.3, TLI was built on top of the old socket code. In the new Sun Solaris OS, this was reversed. The OS primarily intends applications to call TLI, and ‘sockets’ are a patch on top of TLI that was only supposed to be used for backward compatibility with applications that were written to the old socket interface.

But alas, the world is not unfolding as it should. The popularity of the Internet, with its heavy backward dependence on old UNIX standards, means that much relatively free code that uses sockets is being used for new applications. And a lot of information and books on programming using sockets already exists.
Microsoft has adopted their so-called ‘WinSock’ as their transport layer API.

TLI and socket interfaces are very similar so because sockets now seem to be the preferred interfaces for the future, I will teach you the socket interface.

5.6.2 Intro To Sun Sockets

Sun’s Socket Interface is a library of procedure calls, constants, data types, and even the occasional global variable. The data types are mainly for declaring structures for use as procedure parameters; the constants and types are supplied in the header files <sys/types.h> and <sys/socket.h> which you include in that order into your source code.

Though TLI provides state tables which define the order various calls can be made in (e.g. you can’t disconnect unless you’re in connected state), the socket interface is not so well specified.

For an overview of sockets see either:

i) Ch. 10 (“A Socket-Based Interprocess Communications Tutorial”) and 11 (“An Advanced Socket-Based Interprocess Communications Tutorial”) of the Sun Network Programming Manual. See/gfs1/CMPT/371.tront/Sun*.ps


5.6.3 Clients and Servers

The basic architecture of a connection is that there is a caller and callee. The caller is called a client, because it calls up either to get data from a server, or to send data to a data receptor server. Or to do both.

Nonetheless, the server must already exist and be running before the client’s call can be successful (i.e. not rejected because destination port not monitored by a server (“port unreachable” ICMP message)). (When you program a call, you should always check the error code returned for success!) In addition, the client must be made aware of the complete address of the destination port in order to call it.

Sometimes the destination port is not known until the destination/server process starts up and binds to an unused port number suggested by the OS (who maintains the list of assigned and unassigned port numbers). In Sun OS, port numbers between 0 and 1023 are reserved for permanent use (i.e. well-known port numbers for the ftp server, telnet server, etc.). You shouldn’t specify a particular number above 1023 either, as some of them may be in use by other transient processes (like your fellow Cmpt 371 student’s programs)! A transient server (i.e. one whose life only lasts for the duration of the program), should ask for a port number from the OS when it starts up, report that port number to its user. The server user can then make that port number known to clients that want to call that server.

In addition, if the clients don’t know the symbolic name (e.g. gemini.csil.sfu.ca) or network address of the server, that should be printed out by the server program when it first starts up.

5.6.4 Senders and Receivers

Don’t get senders and receivers mixed up with clients and servers. For one way communication, the server could be either a sender (source) or a receiver (sink) of data.

In TLI, a connection is opened for read, write, or read-write mode (same as a file). For sockets, they are always considered two way, though a particular application might choose only to send data one way (e.g. client --> server).

5.6.5 Example Code

I will supply you with some example code which you will modify during Assignment 3. It is located in:

/gfs1/CMPT/371.tront/src/studserv.vX.c
/gfs1/CMPT/371.tront/src/studcli.vY.c

where X and Y are version numbers. Use the latest code only. Note these files use library functions defined in /usr/include/sys/

This tutorial will begin to prepare you to understand the code in these two files, but you will undoubtedly have to do further reading. If you do not understand this code, go back and read the Sun documentation further.

5.6.6 Initialization Procedures

Using sockets, to open a connection to another distant end point (whose full address you know) requires that the other end point already exists (as a server, to provide or receive data), plus you need to make several calls:

1) socket() - to specify the addressing format, the mode (typically only SOCK_STREAM and SOCK_DGRAM are used). socket() returns an “I/O descriptor index” for the opened socket (just like open() returns a file descriptor when opening a file).

All Unix tasks maintain an array of I/O descriptors. The first few elements are used for stdin, stdout, and stderr (i.e. the I/O streams which standard input and output are read from and written to. When you redirect or pipe, these are temporarily modified.)
Extra descriptors entries are initialized whenever you open a file or call socket(). (You can think of the stdin, stdout, and stderr as pre-opened for you).

The return value of the socket() function is just the array index of the newly initialized I/O descriptor. If the socket() call fails, the return value is -1, and global variable ‘errno’ is set to indicate the reason for the failure. The meaning of errno is available from the man page for socket().

Both client and server must call socket() to obtain an I/O descriptor index from the OS.

I have included some of the man pages for the function you will use in this section of the course notes. But if you need to get others, the best way to print man pages is:

```
man socket | col -b | lpr -Pprintername
```

The “col” removes most of the strange formatting (underlines, bold, etc.) for output on a simple printer.

Also, if you want to look at/print out some of the header files, look in `/usr/include/`. 
You probably know what a datagram is, but not the alternative: SOCK_STREAM mode. In contrast to SOCK_DGRAM (which uses UDP), SOCK_STREAM is a connection-oriented mode (which uses TCP).

Many connection-oriented protocols regard the data channel as a stream of bytes. If you put two 1000 byte packets in, the destination may read this as one 2000 byte receipt, or four 500 byte receipts, etc. This depends on whether both 1000 byte packets have both arrived when the receiver first tries to read, and how big a buffer the receive function is supplied with. This is an important point.

Another important point is that stream-mode does NOT provide any delimitation regarding the boundaries:

- between what was last sent and the beginning of the first 1000 bytes,
- between the two packets, or
- between the send of the second 1000 and whatever comes next.

So SOCK_STREAM is both connection-oriented (you have to call connect() before sending), and stream-oriented. This is an unfortunate primitiveness is related to the TCP protocol that often underlies sockets, and the failure of the socket implementation to provide added delineation functionality. There is another mode defined, SOCK_SEQPACKET, which implements delineation but which unfortunately is rarely implemented.

2) To associate a certain socket (i.e. I/O descriptor index) with a transport endpoint/port number on your machine, you use a call to bind().

Note that binding does not associate your socket descriptor with a port number on the distant machine, but with one your machine. Which port and machine you are going to connect to is specified in the connect() call (or in the case of datagrams, the sendto() call).

If you are writing a server program, you MUST call getsockname() to find out the actual port number that your OS bound to your socket to (in order to tell the client your full transport address). But before you call getsockname() you MUST have called bind() to make the association between your socket’s I/O descriptor index and your transport endpoint.

If you are a client, you can call bind(), but do not really have to. This is because you don’t need to find out the port number to be used at your end, as no one will be calling you. For a socket client without a call to bind(), a bind takes place automatically behind the scenes when you first use the connect() function (or for datagrams, the sendto() function). So a client does have a full transport address that is sent to the distant server. The server can then know who’s calling, and know what return address to use!

The problem with doing a bind, particularly for internet inter-machine communication, is that you must pass bind() the address of a complex structure describing your transport address. This structure is defined in <socket.h> as:

```c
struct sockaddr {
    u_short sa_family; /*really addr. format*/
    char sa_data[14];
};
```

The socket address family is really another name for the network domain address format. Certain unsigned short integers have been defined as symbolic constants in <socket.h> that can be used to assign to the sa_family field, e.g.

```c
#define AF_INET   2
#define AF_APPLETALK 16
```

The details of the 14 bytes of sa_data vary from address domain to address domain, so it is best to fill out a more appropriate structure and cast it to a sockaddr. The one we are interested in is:

```c
struct sockaddr_in {  /*for internet*/
    short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};
```

Note that struct in_addr is just the 4 bytes of an internet IP address (e.g. 123.78.54.101). (Actually, it is a union of 3 different ways to access those 4 bytes).
So to call bind, you need this code:

```c
struct sockaddr_in this_addr;
this_addr.sin_family = AF_INET;
this_addr.sin_addr.s_addr = INADDR_ANY;
this_addr.sin_port = 0;
if(bind(socket_descr,
       (struct sockaddr *)&this_addr,
       sizeof this_addr)
       <0) {
    perror("for first bind");
    exit(1);
}
```

Note that setting the 32 bit version of the sin_addr union to INADDR_ANY (which is 0x00000000) just says use (any of) this machine's internet IP addresses. Some machines which bridge between nets have two or more IP addresses.

Setting sin_port = 0 says associate this socket with any free port of the OS's choosing. This is normal procedure as we don't want to bind to a port already in use on this machine by another user.

Note that if bind returns a value < 0, the reason is stored in the global variable errno. perror() is a helpful function that examines errno and prints out an appropriate message plus the programmer's string. Typically you call exit() to halt after an error.

Once the bind is done, you can find out the IP address and port actual bound to, by calling getsockname() with a parameter list almost identical to bind(). By looking into the sockaddr structure returned you can find out the actual local transport address bound to.

Remember, bind() and getsockname() calls are only really required on servers.

### 5.6.7 Network Byte Ordering

Some computers store the least significant byte of an int or long int (or structure or union) lower in memory that the most significant byte, and other brands of computers do the reverse. This really screws networking up. It screws both the network calls like bind(), and it screws up applications. To make sure network code handles this properly, ALL programs should use byte ordering conversion functions. Four are supplied:

- `netlong = htonl (hostlong);`
- `netshort = htons (hostshort);`
- `hostlong = ntohl (netlong);`
- `hostshort = ntohs (netshort);`

For instance, the two byte port number returned by getsockname() should be converted to host (i.e. local) ordering as follows:

```c
bound_port = ntohs(this_addr.sin_port);
```

This implementation of this function does nothing on host machines whose byte ordering is the same as the 'net' ordering. By including this call even on such machines, the resulting code is portable to machines which do not use net ordering!
Sometimes we do not know the network address of the server we are trying to connect with, but we do know it’s name (e.g. sirius. csil.sfu.ca). The function gethostbyname() will return a pointer to a structure with the desired CPU’s 32 bit IP network address buried in it. This structure is very complicated so we will discuss it later. The example code supplied for the assignments also show how it is done.

5.6.8 Connection Phase Procedures

Connecting is only used in SOCK_STREAM mode. In SOCK_DGRAM mode you simply start sending (using sendto() rather than send() because for datagrams you have to specify the destination address in each packet).

Connecting is pretty simple. You simple call connect() with a parameter list similar to bind. Only this time, the second two parameters describe the distant transport address to connect to the local socket specified by the first parameter. The man page for connect is presented in the next 3 pages.
Be CAREFUL to check the return value of connect in your code, because if the server is not present at that address, it will fail. I and most students forget to start the server occasionally and encounter this problem even when testing simple programs.

5.6.9 Data Transfer Phase

Connection oriented data transfer is effected by calls to:

- \( \text{count} = \text{send(socket, &data, length\_data, flags)} \)
- or the standard unix write() which has the same first 3 parameters only.
- \( \text{count} = \text{recv(socket, &buffer, length\_buffer, flags)} \)
- or the standard unix read() which has the same first 3 parameters only.

send() transfers the number of bytes specified by length\_data from the data buffer array out the specified socket. The fourth parameter allows you to specify whether the data should be expedited (if this feature is supported).

send() can block until flow-control back-pressure is received. It is possible to call ioctl() to tell send not to block, thus allowing the program to go on and do other things (like receive acks).

recv() is a blocking read of the specified socket into a specified size buffer. The number of bytes actually read (which will always be \( \leq \) length\_buffer) is returned as the function’s value. -1 is returned if an error occurred. If zero count is returned, that generally means the sender has closed the connection. Note this is similar to using read() and encountering an end of file.

The recv flags can be set to request high priority data, or to peek at the incoming data without removing it from the incoming stream.

The standard Unix read() and write() functions can also be used. This allows code which doesn’t need flags, to be written using standard I/O calls so that, say, stdout could be redirected to a socket.

For datagrams, there are other functions called sendto() and recvfrom() that can be used. Obviously, without a connection, sendto() must have additional parameters to specify the full destination transport address.
5.6.10 Connection Release Phase

The transport layer is reliable. You don’t have to worry about whether the data got through or not (other than total net failure or the complete failure of the destination computer). Thus if you close() a socket, if you have sent data but it has not left the computer because of queuing, the OS will attempt to send it for a long while before giving up. This is nice, but not truly graceful as described below. I would only call it ‘somewhat graceful’ (a Tront’ism).
Note: If you really don’t want the queued portion of outbound data sent, you can see the man page for shutdown(). When using TLI’s t_snddis(), the abort is so abrupt that queued data is not sent.

**Graceful disconnect** is more sophisticated in that it realizes that just because your application wants to stop sending, that doesn’t mean the other end wants to stop.

Most disconnect code is inelegant, because the service definition does not distinguish between 3 situations:

- I don’t want to send anymore, and in fact, may want my out queue cleared too. i.e. a very abortive disconnect.
- I don’t want to send anymore, but at least finish sending what is queued for sending. i.e. a somewhat graceful disconnect. Both abortive types ignore the wishes of the other station.
- I don’t want to send anymore, but will gracefully receive until the other end finishes.

I do notice that Microsoft’s WinSock API has an option for graceful disconnect, but it is really the second ‘somewhat graceful’ version.

Most socket libraries are not very graceful because they were written long ago. In fact, the Americans were so infuriated at ISO for not providing a very graceful kind of close that the ANSI standard for the same service has this added (and is thus incompatible with a world standard).

Also, most socket libraries fail to handle a receiver disconnecting nicely. The send() function does not return any errors indicating that the receiver has abortively closed. I guess the idea is that most receivers are dumb sinks. But what if they run out of file space. Wouldn’t it be nice to somehow allow the send programmer to figure this out. On many Unix systems, a ‘signal’ will be sent to your process. Unfortunately, for that to do any good I believe you have to somehow register with the OS an interest in asynchronous I/O event signals. This is an advanced Unix programming topic.

In TLI, it was possible to disconnect without unbinding. You could thus use your bound port number to receive a number of calls in a row all addressed to the same advertised port number. This is not possible using the socket interface because there is no disconnect() procedure, only close() which disconnects, unbinds and closes the socket. The reason for this is steeped in Unix multi-processing history. If you can receive multiple calls, why should you write code that assumes that the calls will come sequentially (i.e. that you will finish with one before the next one comes).
5.7 Simple Servers

A server process socket must exist, and be in listening mode before a client can connect to the server. In addition the server must have announced its address (or name) and port number to the client, so the client knows what transport address to connect to. For the moment, we will defer talking about address details so as to not complicate the simple steps of listening and accepting a connect requests.

The server indicates its interest in receiving connect requests by calling listen(). The call to listen() includes the local socket descriptor to listen on, and an integer suggesting how long a queue the system should maintain of incoming, but as yet unaccepted connect requests. (I believe that for SunOS, this queue cannot be set larger than 5 in length).

listen() does not block waiting for the first call. As shown on the man page, listen() immediately returns the success or failure of the function (and via errno, a reason for the failure. e.g. invalid socket number or queue size).

To dequeue and use the first connection request requires a call to accept(). accept() is a bit funny as the call is not accepted on the same socket as it came in on. In fact, accept returns the descriptor index of a new socket, already opened and properly bound for you to the incoming connection.

This is like having a corporate receptionist fielding incoming telephone calls on line 1, but forwarding them to the president and telling him the incoming call appears to him to be on line 3. This frees line 1 up to receive another call (which is good, since line one is the only one that answers calls to the corporation’s advertised phone number).

The reason for this design is that Unix (for which the socket interface was originally designed) is a multi-tasking OS capable of receiving more than one connection at once. In fact, receiving processes such as ftp typically spawn extra clone processes to deal with each simultaneous client, connecting with each one on different socket assigned by the accept() function.

Note: We have not yet covered how to have a server fork (instantiate) itself for as many clients as want to simultaneously connect to it. Nor how to send data...
while simultaneously waiting for the odd piece of received data arriving irregularly.

accept() also returns the calling client’s full transport address and the length of the caller’s full transport address (since it may be an Appletalk client calling). It is nice to know who is calling!

5.8 Using the Address Data Structures
Both the socket and TLI transport interfaces use very complicated data structures. This is required for three reasons:

1) To make code portable to machines that use a different CPU design, even though they use the same OS.
2) To make code portable to machines that use a different OS.
3) To make connections to computers with different CPU/OS/protocols.

TLI is probably more complicated in this regard than the socket interface standard.

This subsection will help you understand the structures used to hold full transport addresses, how to read out an IP address from a structure, and how to find out the IP address of a client whose symbolic name (e.g. gemini.csil.sfu.ca) you know.

5.8.1 Getting Your Server’s Transport Address
This can be done with a call to getsockname() once the server has bound itself to a socket.
the server program.

getsockname() returns the sockaddr structure filled in with the full address of the server, and the length of the full address (which may be less than the length passed in).

Recall:

```
struct sockaddr {
    u_short sa_family; /*really addr. format*/
    char sa_data[14];
};
struct sockaddr_in { /*for internet*/
    u_short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};
struct in_addr {
    union{
        struct{u_char s_b1,s_b2,s_b3,s_b4;} S_un_b;
        u_long S_addr;
    } S_un;
};
The first thing you should do is check that address
domain/format/family. This will tell you whether the
address format is internet or, say, Appletalk:
#define AF_INET   2
#define AF_APPLETALK 16
```

If it is for internet, you can cast the sockaddr to a
sockaddr_in structure. This will give you access to the
port and machine IP address. Finally, you can see that
the IP address is of type in_addr. This is really a union
of two different ways you can reference the 32 bits of
the variant record: by individual bytes, or as an
unsigned long.

All these type definitions are found in the header file(s)
I mentioned at the start of the tutorial.

In order to, say, print out the complete transport
address, you must reference down into this structure.
In particular, IP machine addresses are normally
written for humans in the form "127.3.68.107", where
each section is a decimal number < 256 representing
one byte of the machine address.

Some code to do this is included in both the client and
server supplied in:
gfs1/CMPT/371.tront/src/studcli.vX.c, and
gfs1/CMPT/371.tront/src/studserv.vY.c

There is no reason why you can’t dive right down into
this union, but code referencing structures containing
structures containing unions containing structures is
almost unreadable. So I first make a temporary copy
of the IP address in a local variable to eliminate some
of this depth.

```
union{
    unsigned long host_long;
    struct{
        unsigned char b1,
        b2,
        b3,
        b4;
    } host_bytes;
} hostaddr;
hostaddr.host_long = ntohl(myaddr.sin_addr.s_addr);
port = ntohs(theServer.port);
printf("caller's addr was: \n");
printf("AddrFamily=%hd\n", theServer.addr_family);
printf("CPUaddr=%hd.%hd.%hd.%hd\n",
    hostaddr.host_bytes.b1,
    hostaddr.host_bytes.b2,
    hostaddr.host_bytes.b3,
    hostaddr.host_bytes.b4);
printf("port=%hu\n",port);
```

Note: %hd is the C language output format code for
half length decimal (base 10) integers.

This kind of code is used for two reasons in the server.
First, it is used by the server to print out its own
transport address for the client to know where to call.
5.8.2 Who’s Calling?

Second, this same kind of code as above is used by the server to access and print out the caller’s full transport address, as returned in a parameter from accept().

5.8.3 Getting IP Addresses From Symbolic Names

Often you know the symbolic name of the server machine to call (e.g. gemini.csil.sfu.ca), but not it’s 32 bit IP address. It turns out there is a function called gethostbyname() with exactly this purpose.

gethostbyname() takes a point to a character string containing the symbolic name to be looked up in the network database (i.e. Domain Name Service). It returns a complex structure of type hostent.

In the header file <netdb.h>, you will see a #define within the structure that is widely used to access the first of a machine’s several addresses.

```c
struct hostent {
    char *h_name; /* official name of host */
    char **h_aliases; /* symbolic alias list */
    int h_addrtype; /* host address type */
    int h_length; /* length of address */
    char **h_addr_list; /* list of addresses */
    /* from name server */
#define   h_addr  h_addr_list[0]
};
```

By copying the (h_length) bytes referred to by h_addr (i.e. by h_addr_list[0]) to the transport address which the client wants to call, the machine address portion of the destination transport address is specified.

Using a union similar to what we recently saw, you could also print this out in human readable IP form.
5.9 Sophisticated Servers

5.9.1 Handling Multiple Sequential Clients

It is possible with TLI to put a server in a while() loop, where it would handle one connection after another sequentially. It would loop listening, accepting communicating, and disconnecting from a sequential series of callers to the same port. e.g.

```c
    t_open();  /*like the socket() call*/
    t_bind();
    while (1) {
        t_listen();
        t_accept();
        t_snd();
        t_rcv();
        t_snddis();  /* <--disconnect */
    }
    t_unbind();
    t_close();
```

The architecture of this code fragment would not work for the older socket interface, as there is no way to disconnect a connection without also unbinding and closing the socket.

The socket mechanism works a different way. Firstly, in TLI it is t_listen() that blocks waiting for a call. This is why the t_listen() is within the loop above. In contrast, a socket listen() just puts that socket in listen mode with a particular queue size, then immediately returns. It is the socket accept() call which blocks waiting for incoming calls.

Second, the socket accept() call returns the socket descriptor for a new, magically-created socket on which the server program can 'take the call'. This is like the corporate receptionist and president analogy several pages back. Once the server program is finished with this call, it can close the magically-created socket without loosing the original, well known/advertised socket. e.g.

```c
    while (1) {
        magic_socket = accept(listen_socket, 
                              callers_addr, 
                              callers_addr_len);
        if (magic_socket < 0){
            perror("accept failed");
            exit (10);
        send(magic_socket, ...); /* use the */
        recv(magic_socket, ...); /* connection */
    close (magic_socket);
    }
    close(listen_socket);
```

5.9.2 Handling Multiple Simultaneous Clients

The above sequential handling of callers is not very efficient as a blocked operation (even a file read) on one connection holds up the queue of callers that could potentially be being handled by a multi-tasking server.

The TLI t_accept() function has an option similar to sockets where the call is 'taken' on another endpoint. This is the key to both TLI and socket interface servers handling multiple simultaneous clients. Another key is forking a separate process (or thread) to handle each incoming call.

If you don’t already know this from Cmpt 300, a Unix process’s main way of creating another process is actually to create a clone of itself. The clone, which has the same source code, detects it is a clone and branches to a clone-specific part of the server’s code. The server can create extra-copies (or child instances) of itself, one to handle each connection. Though this is a topic for an O.S. course, I will show you this briefly.

Note: It would make much more sense if threads were used, but Unix is process-oriented, not thread-oriented.

The (somewhat inelegant) way which Unix spawns new tasks is a call to the fork() function. fork creates an exact duplicate of the calling task. The only difference is that fork gives a function return value of 0 to the child, vs. > 0 to the calling task (and -1 if the fork failed)! The parent is returned the child’s task ID which is > 0.

So on receipt of a connect indication (i.e. when accept() unblocks), the server should fork itself. The original (parent) server should terminate/close its relationship with the magic socket, and loop back to wait for/accept the next call. If another one calls it forks yet another child to handle it. This is how an FTP server works!

A child, which has the same code as its parent, is born in a running state just returning from the fork. It should terminate/close its relationship with the listen socket, deal with the incoming call on the newly opened magic port by sending/receiving data, close the connection (magic_socket) when done, then die!
while (1) {
    magic_socket = accept(listen_socket, 
      callers_addr, 
      callers_addr_len);

    switch (fork()) {
      case -1: 
        perror ("fork failed");
        exit (20);
      case 0:  /*child executes this case*/
        close (listen_socket); /* not 
        /*interested in listen_socket*/
        send(magic_socket,...);/*use it*/
        close (magic_socket);
        exit (0); /*child dies*/
      case default: continue;
        /* parent expresses disinterest 
           in magic socket*/
        close (magic_socket);
        /* and loops back to wait 
           for next caller */
    }
  }
}
close(listen_socket);

Notes:

- that you don't need an array of magic socket's to handle 
  the many callers, as each forked task has its own 
  complete program and address space (i.e. variables) 
  separate from the parent.
- All of the above may be taking place on one transport 
  layer port.  A socket is not a port. At minimum it is a 4- 
  tuple: a local net address + port number + process 
  number + protocol flavour. And the child and parent 

5.10 Handling Full-Duplex Send and 
Receive

Often, you want to send a large file while 
simultaneously receiving ACKs (i.e. continuous 
ARQ). Stop-and-wait works fine, but if cumulative 
ACKs are sent back only every second or third packet, 
we will block waiting to recv() an incoming ACK 
which may never be sent.

The solution is to call a non-blocking recv() 
between each send packet, to see if any return data or 
acks have arrived. If none has, send the next packet. If 
some has, handle it, then send the next packet!

There are a number of ways to invoke non-blocking. 
In TLI, passing a t_ndelay flag to the t_open() function 
will cause calls concerning that port to not block. With 
the socket interface, you have to call:

fcntl(socket,F_SETFL,FNDELAY)

on an already open socket. This sets the flag 
(sometimes alternately called O_NDELAY) for that 
socket to no delay.

Also on Unix, a call to poll() will find out if any 
'events' (e.g. a packet arrivals) have occurred on a 
particular i/o descriptor, or even on any one in a set of 
descriptors that you can simultaneously be waiting on.

Sometimes if two applications are exchanging data in 
an unpredictable full-duplex manner, sending with non- 
blocking recv()s between each send is not 
acceptable. This is because the send may block due to 
flow control back pressure, thence preventing reception 
of data. How do you handle this?

There are several ways:

1) Use a non-blocking send() too! If both send and 
receive are inactive, this results in a busy loop. Use 
sleep (2) seconds in the loop so as to not gobble up 
CPU time on a multi-tasking OS.

2) Use a blocking poll on the set of two types of 
sockets events. It will unblock if either a packet is 
received or send flow control is released. (i.e sleep 
until interrupted).

3) fork() a separate process for either the send or 
receive tasks so that even if one is blocked the other 
can run.

The above info may not be accurate or applicable on 
your operating system, but illustrates some advanced 
OS concepts taken from several OSs (unix, mach, vms) 
and languages (ada).
5.11 TCP - Transmission Control Protocol

TCP is the one of the oldest and most common transport layer protocols. TCP is normally associated with the Ethernet MAC protocol, the Unix operating system, and the "socket" concept. Also, it normally runs on top of an internetworking sub-layer called IP. Thus you commonly hear the pair referred to as "TCP/IP". But, more and more we are finding these mixed with other systems. e.g. Sockets are showing up as a new MS-Windows A.P.I., and TCP/IP is used on other systems than Unix (after all, internetworking amongst only Unix computers was rather restrictive).

TCP is a connection-oriented, stream-oriented protocol. Once a connection is made, you send bytes, not packets. If you send 1000 bytes, they may arrive in two groups of 500, or 10 groups of 100, etc. Or two 1000 byte packets may arrive as one 2000 byte group. i.e. There is no transport layer "framing" in the stream-oriented sub-class of a connection-oriented transport protocols. It just models a bit pipe stream. Bits put in one end are guaranteed to get thru, or you will be notified.

Since TCP is stream-oriented, its (end-to-end) flow and error control do not work on the concept of packets, but instead each byte is given its own sequence number. These are measured modulo $2^{32}$. Similarly, ACK numbers are modulo $2^{32}$, and are use the 'next byte expected' convention. These acknowledge numbers are of the cumulative type, and take care of lost or duplicated TPDUs. Recently, an option has been added to selectively NACK a range of byte numbers, rather than using Go-Back-N.

TCP also has an optional frame check sequence, which some people feel is necessary since the IP layer just below does not check its payload. But a TCP frame check sequence seems unnecessary because the Ethernet typically found further below at layer 2 uses a strong 32 bit check sequence. But if you are using SLIP instead of Ethernet at layer 2, it would be wise to turn on TCP payload checking.

To further discuss TCP it is best to show you a TCP header (Figure 6-24 from [Tanenbaum96]). One of the unfortunate things about the TCP frame format is that there is not header field to indicate the type of data that the TCP frame is carrying. Is it carrying user data, or does its payload contain yet a higher level protocol? Or does it carry some network layer packets being tunnelled thru this particular net? In Unix, this is sometimes differentiated by well-known port number; for instance if the TCP is carrying File Transfer Protocol (FTP), it is normally sent to port 21.

Figure 6-24 from [Tanenbaum96]

The first long word of a TCP TPDU header is the source and destination port numbers (source port numbers are included so you can know which one to reply to).

The next long word is the 32 bit send sequence number. It indicates the byte number of the first byte of payload measured modulo-4G from the beginning of the connection. The third long word is the byte number (also modulo-4G) you are expecting next (i.e. a cumulative ack).

Next comes a complicated long word, the first field of which is a 4 bit measure of the header length (measured in units of long words). This tells the destination how many option fields there are, and more importantly, where the transport payload actually begins.

Following 6 unused bits, are 6 bit flags:

- URG signals that the payload carries urgent data which should be handled by the receiver process with higher priority than other data. This urgent data begins with the first byte of the payload and contains as many bytes as specified by the urgent pointer field. The urgent pointer can be thought of pointing to the end of the urgent data (normal data could follow the urgent data in the payload). Note: when you call the transport layer from a
program and tell it to send urgent data, this causes an immediate push of the urgent data (to be discussed in a moment).

- ACK signals that the number in the ack field is valid (i.e. this TPDU is actually acknowledging something). If this flag is false, the 32 bit ack field is still present (as the first part of a TCP header is of fixed format), but contains junk.
- PSH signals that this data was pushed by the source application (i.e. it was sent immediately rather than buffered up waiting to see if more source user data could be accumulated before boring to send a TPDU). This is frequently done in Telnet, as you want what the user typed sent within a second or two, and not have the source wait to see if the user will type anything more. On reception, the push flag true indicates that the data should not be buffered upon reception. Blocked calls to recv() should be unblocked right away, even if for just a byte or two that has arrived in a short payload.
- RST indicates a reset request. A port that receives a TCP packet for which it does not have a connection replies with a RST. It is also used to reject a request for connection.
- SYN is used to indicate the TPDU is a connection request or connection accepted TPDU. ACK distinguishes between these two cases. If ACK=0, then it is a connection request and the send sequence number is a randomly chosen number to eliminate any problems with lost or duplicated control packets. If ACK=1, the TPDU is the second of a 3 way connect handshake. The acknowledgement sequence number would be one higher (next number expected) than the random one received.
- FINish is a connection release, and indicates the sender has no more data to send.

End-to-end flow control is handled using a credit allocation technique. The window size indicates how many bytes starting with the next number expected the sender is willing to receive. There is a bit of a problem here as the 16 bit credit only allows 64K to be sent max. This is ridiculously small on high speed, long delay (e.g. satellite) channels. RFC 1323 specifies a option negotiation technique whereby this window size can be considered to be scaled by up to 16 powers of 2. This allows a credit for 4 GB to be sent!

There is a 16 bit frame check sequence, but I don’t know why this would be needed unless both layer 2 and 3 did not do one. If all zeroes, it indicates it is not in use. Otherwise, it is a 1’s compliment checksum calculated over the payload, the header, and a weird pseudo header (composed of the source and destination IP addresses, the protocol field from the IP header, and IP’s calculation of the length of the TPDU). The length of the TPDU is not even a field that can be copied from the IP header (its length field specifies the entire length of the IP packet including the IP header). This weird mixing of stuff from layer 3 into layer 4 is really poor.

In fact, when transmitting, layer 4 has to reach down into layer 3 to find this stuff out to calculate the layer 4 checksum! If IP’s header were not checked I could understand this; a packet might have got to the wrong destination. By everything I read says IP’s payloads are not checked, but their headers containing directly or indirectly the pseudo-header info is checked.

The options field allows various things like window credit scaling, and selective NACKs to be specified. The later would specifying the range of bytes (first and last, or first and length) that have not been received properly. Note: the payload is called a ‘segment’ of the connection’s transfer.

The TCP frame format is not all that required to specify the protocol. In addition you need a call interface that can be used by the client programmer. And you need a finite state machine to carefully specify the actions/reactions appropriate in each mode. Figure 6-28 of [Tanenbaum96] shows the TCP state machine (Page 5-23 of these notes).

For a more information on TCP, an example of implementation code, and related protocol implementations (e.g. IP, ICMP, IGMP, UDP, ARP, RIP, SNMP, and OSPF), see the 3 volume set of books [Commer94]

### 5.12 UDP - User Datagram Protocol

The connectionless transport layer protocol widely used on the Internet is UDP. The UDP header is simply four 16 bit words: source port, destination port, TPDU total length (including the header), and the UDP checksum.

It seems to me that the length field is not needed (there isn’t one in TCP; the underlying network and data link layers delimiters are used). I also feel, as I do for TCP, that a checksum is not needed. But I guess if UDP is being used with absolutely no error checking in layers below (e.g. on SLIP links on old non-error correcting modems), it is a nice option.

Also notice that UDP has the same limitation that TCP has in that it has no header field describing the nature/protocol of the payload. Again, like TCP, the destination port number implies this. After the IP header, the UDP TPDU format looks like:

<table>
<thead>
<tr>
<th>Source Port</th>
<th>Destination Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPDU length</td>
<td>Checksum</td>
</tr>
<tr>
<td></td>
<td>UDP Payload</td>
</tr>
</tbody>
</table>

5-99
5.13 References


Table of Contents:

5. THE TRANSPORT LAYER - Layer 4 .................... 5-1
   5.1 Transport Layer Services ........................................ 5-3
       5.1.1 Handling Lost/Duplicated Packets ..................... 5-3
       5.1.2 Expedited Delivery ........................................ 5-4
       5.1.3 Upward Multiplexing ....................................... 5-4
       5.1.4 Transport Endpoint Addressing ......................... 5-5
       5.1.5 Quality of Service .......................................... 5-6
       5.1.6 Downward Multiplexing ................................... 5-6
   5.2 Transport Difficulties ......................................... 5-8
       5.2.1 Error Control ................................................. 5-8
       5.2.2 Flow Control .................................................. 5-9
   5.3 Transport Connection Management ....................... 5-10
       5.3.1 Duplicate Handling ......................................... 5-16
       5.3.2 Connection Establishment ................................ 5-18
       5.3.3 Connection Close (Disconnect) ......................... 5-23
       5.3.4 Transport CPU Crash ....................................... 5-26
   5.4 Examples Of Transport Protocols .......................... 5-27
       5.4.1 TP4 (ISO 8072-8073) ...................................... 5-27
   5.5 The Transport Software Interface ......................... 5-33
       5.5.1 Transport Addresses ........................................ 5-34
       5.5.2 Transport Communication Phases ....................... 5-35
   5.6 Tutorial On SUN Sockets ....................................... 5-36
       5.6.1 Background ................................................... 5-36
       5.6.2 Intro To Sun Sockets ....................................... 5-37
       5.6.3 Clients and Servers ........................................ 5-38
       5.6.4 Senders and Receivers .................................... 5-39
       5.6.5 Example Code ............................................... 5-39
       5.6.6 Initialization Procedures ................................. 5-40
       5.6.7 Network Byte Ordering .................................... 5-51
       5.6.8 Connection Phase Procedures ......................... 5-53
       5.6.9 Data Transfer Phase ....................................... 5-57
       5.6.10 Connection Release Phase ............................... 5-64
   5.7 Simple Servers .................................................. 5-70
   5.8 Using the Address Data Structures ....................... 5-76
       5.8.1 Getting Your Server’s Transport Address ............. 5-76
       5.8.2 Who’s Calling? .............................................. 5-81
   5.9 Sophisticated Servers .......................................... 5-85
       5.9.1 Handling Multiple Sequential Clients................ 5-85
       5.9.2 Handling Multiple Simultaneous Clients .............. 5-87
   5.10 Handling Full-Duplex Send and Receive .................. 5-91
   5.11 TCP - Transmission Control Protocol .................... 5-93
   5.12 UDP - User Datagram Protocol ............................. 5-101
   5.13 References ....................................................... 5-102

5-106